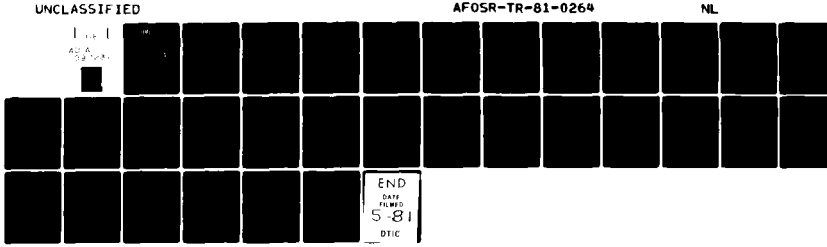
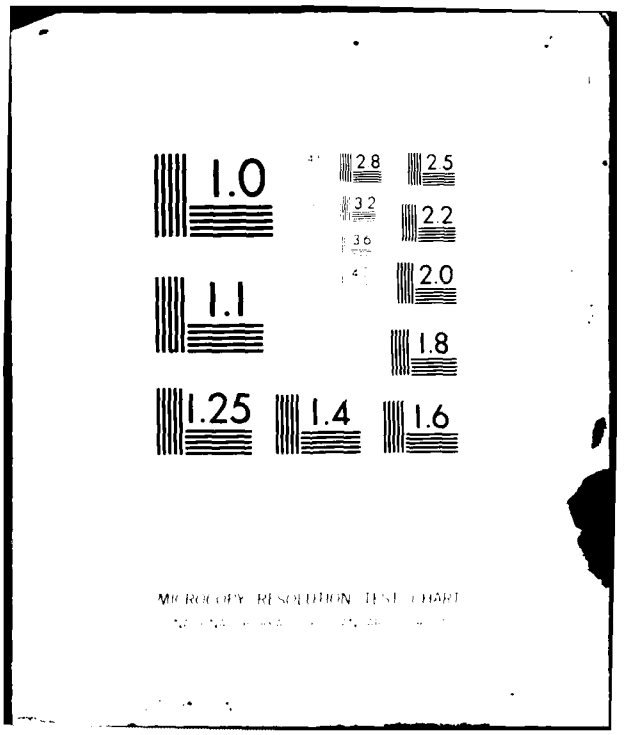
AD-A097 231 MARYLAND UNIV COLLEGE PARK DEPT OF COMPUTER SCIENCE F/G 9/2
EVALUATING A DATA ABSTRACTION TESTING SYSTEM BASED ON FORMAL SP--ETC(U)
DEC 80 P R MCMULLIN, J D GANNON F49620-80-C-0001
UNCLASSIFIED AFOSR-TR-81-0264 NL



END
DATE FILMED
5-81
DTIC


1.0
2.8
2.5
3.2
2.2
3.6
2.0
1.1
1.8
1.25
1.4
1.6
MICROCOPY RESOLUTION TEST CHART

LEVEL II

Evaluating a Data Abstraction Testing System Based on Formal Specifications

Paul R. McMullin

John D. Gannon

Department of Computer Science
University of Maryland
College Park, Maryland

Abstract

A compiler-based specification and testing system for defining data types has been developed. The system, DAISTS, includes formal algebraic specifications and statement and expression test coverage monitors. This paper descibes our initial attempt to evaluate the effectiveness of the system in helping users produce software containing fewer errors. In an exploratory study, subjects without prior experience with DAISTS were encouraged by the system to develop effective sets of test cases for their implementations. Furthermore, an analysis of the errors remaining in the implementations provided valuable hints about additional useful testing metrics.

Key Words and Phrases: data type, experiment, specification, testing.

AIR FORCE OFFICE OF SCIENTIFIC RESEARCH (AFSC)
NOTICE OF TRANSMITTAL TO DDC
This technical report has been reviewed and is approved for public release IAW AFR 190-12 (7b).
Distribution is unlimited.
A. D. BLOSE
Technical Information Officer

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER (19) AFOSR-TR-81-0264	2. GOVT ACCESSION NO. AD-A094 231	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) EVALUATING A DATA ABSTRACTION TESTING SYSTEM BASED ON FORMAL SPECIFICATIONS	5. TYPE OF REPORT & PERIOD COVERED (9) Interim rept.
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) (10) Paul R. McMullin and John D. Gannon	8. CONTRACT OR GRANT NUMBER(s) (15) F49620-80-C-0001 ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS University of Maryland Department of Computer Science College Park, Md. 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS (16) 61102F (17) 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB, Washington, DC 20332	12. REPORT DATE (11) December 1980
	13. NUMBER OF PAGES (12) 31
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approve for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

data type, experiment specification, testing.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

A compiler-based specification and testing system for defining data types has been developed. The system, DAISIS, includes formal algebraic specifications and statement and expression test coverage monitors. This papaer describes our initial attempt to evaluate the effectiveness of the system errors. In an exploratory study, subjects without prior experience with DAISIS were encouraged by the system to develop effective sets of test cases for their implementations. provided valuable hints about additional useful testing metrics.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

1. Introduction

Program development remains an error-prone process. Specifications are often ambiguous or incomplete, and validation of a program's conformance to its specification by testing is often performed by humans who agree too readily with the output of the test program and who have little feel for how thoroughly the specification or program has been tested. These problems are magnified during the later stages of the software life cycle. Since specifications, programs and test data are usually separate entities, each can be altered without regard to the others. Furthermore, test data collected because it exposes a particular error is theoretically useless as soon as the error it exposes is corrected.

Program testing systems have been developed that compare user-supplied and program-computed input-output pairs, and measure several program coverage criteria (e.g., statements, paths, expression values, etc.) [Hamlet 78]. These input-output pairs can be difficult to write for complex functions (e.g., the result of adding a single identifier to a hash-coded symbol table), and when an error is detected, the user-supplied pair is often as suspect as the computed one. Testing systems whose criteria for test data selection involve only program structure are too weak to reveal all design errors and many types of construction errors [Goodenough and Gerhart 75].

We have combined recent work in data abstraction specification and modularization with a program testing system in an attempt to ease program development. DAISTS (Data Abstraction Implementation, Specification, and Testing System) [Gannon, et al. 80] combines a data abstraction language containing SIMULA-like classes [Dahl, et al. 68] and algebraic specifications similar to those of [Guttag 77] with a library of test monitoring routines. With user-supplied test sets, the axioms of the specification are used as driver programs for the

implementation. Structural testing criteria are applied to both axioms and code to evaluate the test data. We feel DAISTS has several advantages over conventional program development systems:

1) The specification, program, and test data are packaged as a single entity, encouraging their mutual maintenance.

2) The specification language is applicative and the implementation language is imperative. We hope that this orthogonality will reduce the liklihood of the same error appearing in both the specification and the implementation.

3) The test data coverage of the specification and the program are measured.

4) There is no need to describe the concrete representation produced by an operation; the user (specifies and) writes an equality routine to judge the results of tests for abstract objects. This simplifies the testing process by removing the requirement for 'hand simulation' of complicated operations.

5) Having a tool that incorporates specifications into the development process should provide the motivation and experience necessary for programmers to use formal specifications effectively.

The construction of every software tool should include an evaluation of its effectiveness. The evaluation can be used to convince users of a system's worth and can also provide useful information to the designers about its shortcomings. This paper describes an exploratory study that compared program development with DAISTS against more conventional programming techniques. We felt that structure imposed on the programming process by DAISTS would aid programmers in constructing programs containing fewer delivered errors, but were concerned about program development costs and the ability of users to adapt to DAISTS.

2. DAISTS

A program submitted to DAISTS contains an implementation of an abstract data type written in the high level language SIMPL-D [Gannon and Rosenberg 79], a collection of algebraic axioms describing the type, and a collection of test cases.

2.1. Implementation Language

SIMPL-D class declarations define new types which may subsequently be used in object declarations. The interior of a class is a series of variable declarations (the representation) followed by a series of procedure declarations (the body). Including the names of procedures in the class heading makes the procedures visible outside the class. The appearance of the reserved word assign in the operation list enables the assignment operation (:=) to be applied to objects of this class type.

When a unit of scope containing the declaration of a class object is executed, a new copy of the class object is allocated and initialized. Users generally view class objects as indivisible entities; the components of class objects cannot be accessed outside of its class declaration. Inside the class, these objects may be viewed as structures containing more primitive objects. Statements in the body of a class can access the unique components of a class object using a period notation similar to that of PL/I or Pascal.

A fragment of an implementation for bounded lists of integers is shown below. Each list object is represented by a boolean variable (Empty) indicating whether or not the list is empty, an array of integers (Values) holding the values currently in the list, and integer indices (Head and Tail) into the array identifying the first and last elements.

```
class List = NewList, AddFirst, DeleteFirst, DeleteLast,
             ListLength, ..., assign

   /* The representation */

   define ListSize = "11"
   unique boolean Empty
   unique int Head, Tail
   unique int array Values(ListSize)

   /* The operations */

   List func NewList /* returns list with no elements */
      List Result
      Result.Empty := true
      return(Result)

   List func DeleteFirst(List X) /* returns Tail(X) */
      List Result
      Result := X
      if Result.Empty or Result.Head = Result.Tail
         then /* list contains zero or one element */
            return(NewList)
         else /* increment Head modulo ListSize */
            if Result.Head <> ListSize - 1
               then
                  Result.Head := Result.Head + 1
               else /* circularize to zero origin */
                  Result.Head := 0
               end
            return(Result)
         end

   .
   .
   .
endclass
```

2.2. Specification Language

The specification language for DAISTS is similar to that described in [Guttag et al. 78]. The primitives of the language include boolean and integer constants, free variables, equality, other boolean and integer operators, and functional composition. Axioms equate two expressions; the first expression is generally a function composition and the second expression is a combination of primitives and conditional expressions like those of ALGOL 60. Each axiom presented to the DAISTS processor is named and has a list of the names and types of the free variables used in the axiom. The axioms for the bounded list operation DeleteFirst might look like:

```
Axioms

DeleteFirst1:
     DeleteFirst(NewList) = NewList;

DeleteFirst2(List AxList1,INT AxInt1):
     DeleteFirst(AddFirst(AxList1,AxInt1)) =
          if ListLength(AxList1) = 11
               then DeleteLast(AxList1)
               else AxList1;
```

The axiom DeleteFirst1 specifies that deleting the first element in an empty list results in an empty list. DeleteFirst2 specifies that the result of deleting the first element from a list to which an element had been added at the head is the same as either: 1) deleting the last element from the original list if the original list was full (because adding an element to the front of a full list caused the last element to be discarded), or 2) the original list if the original list was not full (i.e., the element just added was the one deleted).

2.3. Test Data

The testpoints section of a DAISTS program looks like a procedure, complete with declarations and executable statements. This section allows users to build objects to be referenced in the subsequent testsets section of the program. An object that is expensive to construct can thus be used in testing several axioms without repeating its construction.

The testsets section of the program contains a list of axiom names with values to be substituted for the free variables of the axioms. Sample testpoints and testsets sections for bounded lists of integers are shown below.

```
testpoints
int I
List P1,P2

P1 := AddFirst(AddFirst(NewList,3),4)
P2 := NewList  /* Initialize P2 to be empty */
I := 0
while I < 11
  do
     P2 := AddLast(P2,I) /* Fill P2 up! */
     I := I + 1
  end

testsets
DeleteFirst2: (NewList,3), (P1,5), (P2,7),
              (Conc(P1,P2),44);
```

In the testpoints section, the two lists P1 and P2 are constructed; P1 is (3, 4) and P2 is (0, 1, ..., 10). In the testsets section, test data pairs are provided for the axiom DeleteFirst2. Since the axiom DeleteFirst1 has no free variables, no input data can be supplied for it, and DAISTS will generate one test to see if the implementation holds for it. The four pairs of test data for DeleteFirst2 cause DAISTS to generate tests for this axiom first with AxList1 instantiated as NewList and AxInt1 as 3, then with AxList1 as P1 and AxInt1 as 5, etc. For each test set, the implementation is used to evaluate the left hand side of the axiom, and then again to evaluate the right hand side of the axiom, and then the two values are compared using the standard equality operator for basic types of the language and a user-supplier equality operation for the abstract types. An error message is generated if the two sides do not agree. A restriction placed on the implementations for DAISTS (but not inherent in SIMPL-D) is that the abstract operations have to be functions without side effects so that the two evaluations in an axiom do not modify the test set elements.

2.4. Run-time Monitors

DAISTS also generates code to monitor statement and expression execution at run time. All statements in the implementation and all parts of the axioms must be executed to

have a successful test. Furthermore, all expressions in the statements and axioms must take on more than one value; user-supplied equality operations are used by the system to determine if objects with user-defined types change values. Unexecuted statements or axiom fragments, and constant expression values indicate that either simpler programs or statements can be written or more test data needs to be added to justify the program's complexity.

3. Methodology

3.1. Overview

We hypothesized that testing with formal specifications would reduce the number of delivered errors without increasing the cost of program development. To test our hypotheses, we conducted an experiment where an intermediate class in programming languages was divided into two groups, given identical English language descriptions of the abstract type 'list-of-integers', and assigned to produce implementations of the type in (the same) high level language. One of the groups used algebraic specifications to test and debug their implementations, and the other group used the more traditional debugging method of test programs. The axiom group was supplied with the axioms for the type; the control group was supplied with a test driver that used the abstract operations to sort groups of integers and were allowed to produce other test drivers at their discretion. Both groups had to develop their own test data. At the end of the experiment, we examined the projects that were turned in to discover residual errors.

There are several design decisions that led us to this particular experiment. In evaluating DAISTS there are really two issues to be resolved: the ease with which users could write axioms and the ability of users to develop programs from the axioms. Given the relative inexperience of our subjects (primarily sophmores and juniors), we concentrated on program development rather than specification development hoping to justify a later, more complex experiment with more sophisticated subjects. Obviously, if the program development task proved too difficult, the system's worth would be questioned because writing formal specifications before development would only make the process more difficult. Another problem concerned the materials to be given to the subjects. Providing the axioms to one group

and requiring the control group to devise their own testing programs seemed to make the control group's task more time consuming. Thus, we decided to provide the control group with a test routine (the sort program), which tested nine of the twelve list operations. Since some subjects would undoubtedly test with only the sort routine while others would write their own test drivers (at least for the three untested functions), we would be able to get more detailed information about the behavior of the control group.

3.2. Choosing the groups

The class (of 79 students) met together for lecture twice a week, and was divided into four smaller groups that met once a week with one of two teaching assistants. Two of the small groups (one from each assistant) were combined to form the axiom group (45 students), and the other two groups formed the control group (34 students). Five of the control group students, and four of the axiom group students did not turn in any project, and were dropped from the study. Two more of the control group students, and one of the axiom group students were dropped because their projects did not appear to be independently developed. Despite the fact that the students were warned several times that every compilation was being recorded and that they were required to work independently, three pairs of projects are so (remarkably) similar that we could not objectively consider them to be independent efforts (either they started from decks that were equivalent, and were jointly developed, or one deck started 'development' after the other was completed, with 'development' consisting of a uniform substitution of names and reordering of code segments). One of each pair of the 'non-independent' projects (the 'dependent' one if determinable) has been omitted. This left 40 students in the axiom group, and 27 students in the control group.

Analysis of the grades that they received for the semester showed only one statistically significant difference between the two groups - the control group had slightly higher examination scores. (See Table 0 below.)

Table 0 Group Differences.

		Axiom Group	Control Group
Letter grade	Mean	2.44	2.54
	St dev	.90	.94
	Level	< 80%	
Project grade	Mean	120.8	114.1
	St dev	17.8	26.8
	Level	< 44%	
Exam Grade	Mean	41.00	45.81
	St dev	12.12	6.90
	Level	< 7%	

Letter grades on a scale 1=D, 2=C, etc.
Project grades (five projects not counting experiment) on a 150 point scale.
Composite exam grades on a 100 point scale.

Significance levels for a two-tailed test.

3.3. The Project

This was the first exposure to "encapsulated types" for nearly all of the students in the class, so we chose to assign a rather small project (approximatly 150 lines) to implement "bounded list-of-integers." All of the operations on lists were described in English in the project handout (see Appendix I), which also contained instructions for using the appropriate processor. The subjects were told that they were to implement all of the functions described in the handout and present results demonstrating that the sort routine or axioms executed with no obvious errors.

A manual describing the implementation language and giving other specific information (using the axioms for the axiom group members, writing driver programs for the control group) was also distributed. The axioms and the test driver program that were provided are in Appendices II and III respectively.

Since the test data had to be submitted along with the program at compilation for the axiom group (to allow DAISTS to generate its test driver), we felt that the control group should also be required to submit their test data with their compilation requests to make the development environments more nearly equivalent.

We also provided separate lecture and lab meetings for the two groups, and tried to exchange experimenters so that each group met with each experimenter to nullify any bias our lectures could be giving. The students were informed that they had been divided into two groups, and were asked not to exchange information about how the two groups were different. However, since the abstract type that they were implementing was the same for both groups, some of the details of the implementation language were discussed with both groups present.

3.4. Data Collection

A special processor was set up to limit access to DAISTS - it did not allow members of the axiom group to write separate test drivers, and it did not allow the members of the control group to use the axioms. This processor also saved a copy of a every deck submitted. DAISTS was hidden so that the students were forced to go through the processor (so that all submissions could be recorded), and the students were told that their decks were being collected. This approach led to a number of identical decks being saved by the submission processor - a student would run a deck at his terminal and then run the deck again to generate a listing on the printer, or programs failed to complete execution before their system default time limit was exhausted so

the decks were resubmitted with larger time limits.

3.5. Identifying errors

After all of the students' projects were collected and graded, the files of decks were examined. For each student, the deck that corresponded to the listing that was turned in was separated into the class definition and the debugging data. We then debugged each implementation, both by "desk checking" and by using the DAISTS system, with a large variety of data points for each axiom. Many of the errors that we found were detected by turning on the subscript-checking feature of the compiler - apparently very few of the students used this feature.

As we were debugging their implementations, we found that several student's implementations had 'subjective restrictions' that were not clearly specified in the project assignment. These 'subjective restrictions' could be interpreted as errors, but a case could be made for allowing them as correct restrictions. One student stored only single-digit positive integers. Several students could correctly store any integers except zero, which they used as markers in their representations.

When these 'subjective errors' were counted, the results were not substantially different from the results reported below, which come from only counting the 'objective errors' - code that fails no matter how favorable the input values selected.

3.6. Measuring errors

We also faced a dilemma in chosing which errors to count, and how to report them. We feel that the most conservative objective measure that we can use is <u>functions containing errors</u>. If a function in the submitted project had to be changed, regardless of the number of changes that had to be made to correct the function, it was counted as a single 'function containing an error.' We like the resolution of this measure,

because 1) it is more nearly representation-independent than any error measure that is influenced by the structure of the code of the implementation, and 2) the project description defined functions, the axioms specified functions, and the sort routine used the specified functions.

We compared our measure to that of [Gannon 77] who reported distinct errors and error occurrences where (for example) if the same error in computing the length of a list was made in three places, it would count as one distinct error, but three error occurrences. Our data produced similar results for both of these measures and for the measure 'functions containing errors.'

Several of the students made errors in the selection of their representations. One student used a circular list and had an ambiguity in his representation so that a full list was not distinguishable from an empty one. This error could only be fixed by adding a word to his representation and repairing many of the functions. Another student's project was corrected by merely changing the size of an array in his representation (no functions needed changing). These decks were charged with one incorrect function to account for the change to the representation (in addition to the incorrect functions that they were charged).

In the results reported below, the measure functions that contained objective errors was used.

3.7. Measuring Cost of Development

It is inherently difficult to measure programmer effort. It is especially difficult to measure effort of students who do not work regular schedules and who are not inclined to keep track of efforts on a project near the end of a semester. Since the only enforcable metric which we could employ was the number of runs submitted, and since the previously mentioned duplicated decks involved none of the debugging effort which we were trying to measure, the most convenient and consistent measure that we can

use is number of distinct runs.

3.8. Statistical techniques

We chose to use the Mann-whitney U-test [Siegel 56] for doing the analysis of the data from our experiment. The Mann-Whitney U-test is non-parametric, and our data is (at best) an ordinal measure of performance. Parametric tests also require that the samples be drawn from an uniform distribution, which we cannot guarantee for our data.

4. Results

We report the data both for the subgroups that successfully completed the project and for the entire groups. By successfully completing the project, we mean only that the output of the program that was submitted for grading displayed no obvious errors. Many of the students turned in projects that they knew were not correct - students in the axiom group had error messages complaining about inconsistent axioms, and students in the control group turned in projects for which the sort routine would not correctly sort the integers that they used to demonstrate that their programs worked. In the axiom group, 32 out of 40 who turned in the project successfully completed it, and in the control group 22 out of 27 were successful.

We have subdivided the successful control group into two groups for further comparison: one that wrote and ran small test driver programs in addition to running the sort program (which is more like an integration test than a driver), and another group that used the sort routine exclusively for testing. Of the 22 successful control group members, 7 wrote their own driver programs.

Since the sort program tested 9 of the 12 list operations, we have reported both the number of incorrect functions tested by the sort routine and the total number of incorrect functions. The subjects were required to implement all the functions and were encouraged to write extra functions to display list objects as a debugging aid.

In the tables below, we report the means and standard deviations (following the means in parentheses) of the number of incorrect functions tested by both the axioms and the sort routine, the total number of incorrect functions, and the number of distinct runs.

4.1. All Subjects

All subjects in both groups had similar numbers of incorrect functions tested by the sort routine and distinct runs. While the means favored the members of the axiom group (an average of .18 fewer incorrect functions out of the 9 functions tested by the sort routine and .71 fewer distinct runs), the differences were not significant. As expected, the axiom group did significantly better than the control group in eliminating errors in all the functions.

Table I All Axiom (40) and Control (27) Subjects

	Axiom group	Sort group	Level
Incorrect sort functions	.60(1.18)	.78(1.64)	
All incorrect functions	.82(1.51)	1.78(2.20)	<.003%
Distinct runs	11.77(5.65)	12.48(8.53)	

4.2. Successful Subjects

When we consider only those students who successfully completed the project, the axiom group did marginally better than the control group even on the functions tested by the sort routine. This result appears despite the fact that the sort routine did do a fairly good job of exposing the errors in these routines for those subjects choosing good sets of data to use with it (in the sense that when data for the sort program contained all of the boundary cases of the sort routine, all of the boundary cases of the list functions were tested). Of course, the results are even more striking when we consider all functions that were assigned. The axiom group delivered more correct functions than did the control group while taking fewer runs.

Table II Successful Axiom (32) and Control (22) Subjects

	Axiom group	Sort group	Level
Incorrect sort functions	.12(.33)	.23(.42)	<20%
All incorrect functions	.19(.39)	1.23(1.20)	<.004%
Distinct runs	10.97(5.60)	11.41(7.60)	

4.3. Subjects Writing Their Own Drivers

We expected that those subjects in the control group who wrote driver programs in addition to using the sort routine would test as effectively as the axiom group, but would require more runs to debug their own drivers. The data in Table III supports these hypotheses, except when we consider all the functions assigned. Even those subjects writing their own driver programs did not produce as many correct functions as the axiom group did.

Table III Successful Axiom (32) and Driver-Writing (7) Subjects

	Axiom group	Driver group	Level
Incorrect sort functions	.12(.33)	.14(.35)	
All incorrect functions	.19(.39)	.57(.49)	<2%
Distinct runs	10.97(5.60)	15.14(5.82)	<4%

Examining the runs of the driver-writing subjects to determine why their efforts did not match those of the axiom group, we find a distinct lack of testing discipline. Five of the 7 subjects had test drivers that could exercise all the functions (one subject missed one function and the other subject missed two). Four of the subjects used effective tests, trying a variety of objects in different operations, while while two other subjects with extensive test drivers just did not seem to use enough data to cover the necessary cases. Four of the subjects used drivers before using the sort routine seriously as an integration test. (I.e., they may have used it to compile their implementations initially, but did not try to debug using it.) Two other subjects used drivers only in response to specific errors that occurred in debugging with the sort routine.

4.4. Subjects Testing with the Sort Program Only

Those subjects testing only with the sort program used an average of 1.3 fewer runs than did the members of the axiom group, but the axiom group, but did not produce as many working functions even when we consider only the functions tested by the sort program (8.73 to 8.88). Part of the explanation for this result may be that the sort program did not encourage the members of the control group to test more thoroughly. The sort program's effectiveness as a testing vehicle was impaired by poor selections of test data that did not include the boundary cases of the sort's domain (e.g., empty lists, lists with duplicate members, etc.).

Table IV Successful Axiom (32) and Sort-Only (15) Subjects

	Axiom group	Sort group	Level
Incorrect sort functions	.12(.33)	.27(.44)	<14%
All incorrect functions	.19(.39)	1.53(1.31)	<.002%
Distinct runs	10.97(5.60)	9.67(7.70)	<8%

5. Conclusions

We have shown that DAISTS can encourage even inexperienced users to develop effective tests for their implementations. Those subjects who used only the sort program to test their implementations stopped testing too soon because the data they fed the sort program did not expose errors in their list implementations. The axiom group needed more runs to satisfy DAISTS, but correctly developed more of the functions used by the sort program.

The discipline of testing with DAISTS can help users avoid less systematic testing methods. Even if we consider only the subjects in our study who wrote their own test drivers, we observe a variety of questionable testing practices - omitted functions, failures to consider boundary cases, and generally insufficient test data. The formal specification required by DAISTS identifies the boundary cases and clearly defines their treatment. Furthermore, DAISTS *run-time* monitoring routines ensure that the code handling boundary cases is exercised.

Performing this type of study can also give us insights that help us improve our system. We were frustrated by the ambiguity that the students read into our careful English descriptions (the 'subjective errors' that we identified - single digit integers, using zero for a marker, etc.), but such imprecision is inherent in informal specifications. Even including formal specifications for the subtypes used in building the new type does not prevent the omission of test data that exposes the confusion. We feel that DAISTS-like systems might expose these errors with special-values testing strategies [Howden 78], e.g., adding test sets that include the constant functions of the subtypes (0 for ints, null and blank strings, NewList, EmptyStack, etc.) and the constants of the subtypes that appear in the text of the implementation.

This experiment did not evaluate the subject's ability to write specifications. While many programmers might have difficulty producing axioms without training, we feel that this fact does not render the tool useless. Our own experience in teaching programmers to write algebraic axioms leads us to conclude that they are not as cumbersome as many believe. Another experiment is needed to test the validity of this hypothesis.

The computer science community has reached a consensus on the desirability of requirements analysis and formal specifications. Having a tool which can incorporate specifications into the development process will provide the motivation and experience necessary to use them. Writing formal specifications need not be considered overhead if they can reduce the effort needed to write and debug test driver programs.

6. Acknowlegments

This research was supported by the Air Force Office of Scientific Research (Contract F49620-80-C-0001). Computer support was provided by the University of Maryland Computer Science Center. We would also like to thank Dr Richard Hamlet and Dr Mark Ardis for their contributions to the development of DAISTS.

7. Bibliography

[Dahl, et al. 68]
O.-J. Dahl, B. Myhrhaug, and K. Nygaard: "The SIMULA 67 Common Base Language", Norwegian Computing Centre, Forskningsveien 1B, Oslo 3, 1968.

[Gannon & Rosenberg 79]
John D. Gannon and Jon Rosenberg: "Implementing Data Abstraction Features in a Stack-based Language", Software - Practice and Experience, vol 9, pp 547-560, 1979.

[Gannon 77]
John D. Gannon: "An Experimental Evaluation of Data Type Conventions", CACM, vol 20, no 8, pp 584-595, August, 1977.

[Gannon, et al. 80]
John D. Gannon, Paul R. McMullin, and Richard G. Hamlet: "Data Abstraction Implementation, Specification, and Testing", (submitted for publication), 1980.

[Goodenough & Gerhart 75]
John B. Goodenough and Susan L. Gerhart: "Toward a Theory of Test Data Selection", IEEE TSE, vol SE-1, no 2, pp 156-173, June, 1975.

[Guttag 77]
John V. Guttag: "Abstract Data Types and the Development of Data Structures", CACM, vol 20, no 6, pp 396-404, June, 1977.

[Guttag et al. 78]
John V. Guttag, Ellis Horowitz, and David R. Musser: "Abstract Data Types and Software Validation", CACM, vol 21, no 12, pp 1048-1064, December, 1978.

[Hamlet 78]
Richard G. Hamlet: "Testing Programs with the Aid of a Compiler", IEEE TSE, vol SE-3, no 4, pp 279-289, July, 1978.

[Howden 78]
William E. Howden: "An Evaluation of the Effectiveness of Symbolic Testing", Software - Practice and Experience, vol 8, pp 381-397, 1978.

[Siegel 56]
Sidney Siegel: Nonparametric Statistics for the Behavioral Sciences, McGraw-Hill, New York, 1956.

Appendix I - first page of handout

You are to write the CLASS implementation for lists of integers. A list is an ordered collection of elements which may have elements added and deleted at its ends, but not in its middle. The operations that you must "export" are: AddFirst, AddLast, Conc, DeleteFirst, DeleteLast, First, IsEmpty, ListEqual, ListLength, NewList, and Reverse. Each operation is described in detail below.

The lists are to contain up to eleven (11) elements. If an element is added to the front of a "full" list (one containing eleven elements already), the element at the back of the list is to be discarded. Elements to be added to the back of a full list are discarded. Requests to delete elements from empty lists result in empty lists, and requests for the first element of an empty list results in zero (0).

Remember that the operations that you implement are to be functions, and that they may ***NOT*** change their parameters! If a function needs to manipulate a parameter to perform the operation, the parameter is to be COPIED to a LOCAL variable BEFORE the change is performed! You may use any representation you choose to implement your lists. The detailed operation descriptions are below:

List FUNC Addfirst(List L,INT I) - Returns the list with I as its first element followed by all of the elements of L. If L is "full" to start, L's last element is ignored.

List FUNC Addlast(List L,INT I) - Returns the list with all of the elements of L followed by I. If L is full to start, I is ignored.

List FUNC Conc(List L1,List L2) - Returns the list made up of the elements of list L1 followed by the elements of L2. If L1 and L2 together contain more than eleven (11) elements, then the extras are to be ignored.

List FUNC Deletefirst(List L) - Returns the list containing all but the first element of L. If L is empty, then it returns an empty list.

List FUNC Deletelast(List L) - Returns the list containing all but the last element of L. If L is empty, then it returns an empty list.

INT FUNC First(List L) - Returns the first element in L. If L is empty, then it returns zero (0).

INT FUNC Isempty(List L) - Returns one (1) if L is empty, zero (0) otherwise.

INT FUNC Listequal(List L1,List L2) - Returns one (1) if the two lists are element for element equivalent (e.g. First(L1) = First(L2),...), and zero (0) otherwise. Note that two empty lists are considered equal.

INT FUNC Listlength(List L) - Returns the count of elements in L. An empty list has a count of zero (0) elements.

List FUNC Newlist - Returns a list initialized to be empty

List FUNC Reverse(List L1) - Returns a list containing the elements of L1 in reverse order.

Appendix I - second page of assignment for control group

A test routine has been written for you, or you may write your own test routines. The provided routine reads in groups of integers, sorts them, and prints out the smallest 11 of each group. The test routine expects the groups of integers to be separated by zero. A sample test run using the provided test routine is shown below:

```
@add simpld*project.setup              <done once per run>

@$.SIMPLD,S                            <calls the compiler,
                                          asks for listing>
<list implementation>                  <your CLASS for lists>
$TEST                                  <causes the test routine
                                          to be provided>
<groups of integers, separated by zeros>
@eof                                   <end of the data>
```

The data for the test routine may have any number of integers or groups of integers per card, with the integer 0 separating each group. Spaces are used to separate the integers when more than one integer is on a card.

A sample run for using your own test routine is shown below:

```
@$.SIMPLD,S                            <call compiler as above,
                                          assuming setup is done>
<list implementation>
<your test driver>
$DATA
<your data>
@eof
```

You will be required to submit your list implementation via the deck submission processor (to be discussed in class), and you will also turn in a listing of a run using the provided test routine, that shows several groups of integers correctly sorted.

Appendix I - second page of assignment for axiom group

Axioms have been written which you must use to debug your CLASS. You may add axioms of your own at your discretion. A sample run is shown below:

```
@add simpld*project.setup          <done once per run>

@$.SIMPLD,S                        <calls compiler, asks
                                       for listing>
<list implementation>              <your CLASS for lists>
$AXIOMS                            <causes axioms to be provided>
<your optional axioms>
TESTPOINTS
<your testpoints>
TESTSETS
<your testsets>
START
@eof                                <that's all you need>
```

You will be required to submit your list implementation via the deck submission processor (to be discussed in class), and you will also turn in a listing of a run using the provided axioms, with no axiom failures and all statements executed.

Appendix II The Axioms supplied to the axiom group

```
Axioms

/* These axioms are constructed following the Guttag
   rules for deciding which axioms need to be constructed.
   The functions in the "0" group are:
       IsEmpty, ListEqual, ListLength, First
   The functions in the "TOI1" group are:
       NewList, AddFirst
   The functions in the "TOI2" group are:
       AddLast, DeleteLast, DeleteFirst, Conc, Reverse
*/

IsEmpty1:
   IsEmpty(NewList) = 1;

IsEmpty2(List AxList1,int AxInt1):
   IsEmpty(AddFirst(AxList1,AxInt1)) = 0;


ListEqual1:
   ListEqual(NewList,NewList) = 1;

ListEqual2(List AxList1,int AxInt1):
   ListEqual(NewList,AddFirst(AxList1,AxInt1)) = 0;

ListEqual3(List AxList1,int AxInt1):
   ListEqual(AddFirst(AxList1,AxInt1),NewList) = 0;

ListEqual4(List AxList1,List AxList2,int AxInt1,int AxInt2):
   ListEqual(AddFirst(AxList1,AxInt1),AddFirst(AxList2,AxInt2)) =
      if AxInt1 <> AxInt2
        then 0
        else
          if ListLength(AxList1) = 11
            then /* Need to trim the end off! */
              ListEqual(DeleteLast(AxList1),DeleteLast(AxList2))
            else /* Compare them just as they are! */
              ListEqual(AxList1,AxList2);


ListLength1:
   ListLength(NewList) = 0;

ListLength2(List AxList1,int AxInt1):
   ListLength(AddFirst(AxList1,AxInt1)) =
      if ListLength(AxList1) = 11
        then 11
        else 1 + ListLength(AxList1);


First1:
   First(NewList) = 0;

First2(List AxList1,int AxInt1):
   First(AddFirst(AxList1,AxInt1)) = AxInt1;


/* Now for the "TOI2" function definitions: */

AddLast1(int AxInt1):
   AddLast(NewList,AxInt1) = AddFirst(NewList,AxInt1);

AddLast2(List AxList1,int AxInt1,int AxInt2):
   AddLast(AddFirst(AxList1,AxInt1),AxInt2) =
      AddFirst(AddLast(AxList1,AxInt2),AxInt1);


DeleteLast1:
   DeleteLast(NewList) = NewList;

DeleteLast2(List AxList1,int AxInt1):
   DeleteLast(AddFirst(AxList1,AxInt1)) =
```

```
        if IsEmpty(AxList1)
          then NewList
          else
            if ListLength(AxList1) = 11
              then AddFirst(DeleteLast(DeleteLast(AxList1)),AxInt1)
              else AddFirst(DeleteLast(AxList1),AxInt1);

DeleteFirst1:
   DeleteFirst(NewList) = NewList;

DeleteFirst2(List AxList1,int AxInt1):
   DeleteFirst(AddFirst(AxList1,AxInt1)) =
      if ListLength(AxList1) = 11
        then DeleteLast(AxList1)
        else AxList1;


Conc1(List AxList1):
   Conc(NewList,AxList1) = AxList1;

Conc2(List AxList1,List AxList2,int AxInt1):
   Conc(AddFirst(AxList1,AxInt1),AxList2) =
      AddFirst(Conc(AxList1,AxList2),AxInt1);


Reverse1:
   Reverse(NewList) = NewList;

Reverse2(List AxList1,int AxInt1):
   Reverse(AddFirst(AxList1,AxInt1)) =
      if ListLength(AxList1) = 11
        then AddLast(Reverse(DeleteLast(AxList1)),AxInt1)
        else AddLast(Reverse(AxList1),AxInt1);
```

Appendix III The sort routine given to the control group

```
proc Main /* The driver for the sort program to test lists. */

/* Read in a series of numbers that ends with zero, sort them, */
/* and then print out the smallest 'ListSize' of them. */

int Holder /* A place to read numbers into */
int Setnumber /* The number of the current set */
int Counter /* A counter of the number of numbers in Unsorted */
List Unsorted /* Where the unsorted numbers are read into */
List Sorted   /* Where the sorted numbers are stored! */

/* Main loop for reading in sets of numbers */

Setnumber := 1 /* Start to work on the first set */
while .not. eoi do
     Sorted := NewList /* No sorted numbers in this group */
     Unsorted := NewList /* Also no unsorted numbers in yet! */
     read(Holder)              /* To initialize Holder! */
     while Holder <> 0 .and.
          .not. eoi do     /* Keep reading and sorting */
          Counter := 0   /* Set to count the unsorted list */
          while Holder <> 0 .and.
               Counter < ListSize .and. .not. eoi do
                    Unsorted := AddFirst(Unsorted,Holder)
                    Counter := Counter + 1
                    read(Holder) /* Get the next number of the set */
            end

          /* Either Unsorted is full, or this set is finished! */
          /* Must first join unsorted numbers with sorted ones */

            Sorted := Merge(Sorted,Sort(Unsorted))
       end

     /* Here we must have hit an end of a set! */
     /* Print out the first 'ListSize' worth of numbers */

     write(skip,'Sorted numbers of set number',Setnumber)
     while .not. IsEmpty(Sorted) do
          write(First(Sorted)) /*Output smallest number in set*/
          Sorted := DeleteFirst(Sorted)
       end
     Setnumber := Setnumber + 1 /* Now start the next set */
  end
write(skip,skip,skip,'Out of sets of numbers to sort')


List func Merge(List M1in,List M2in) /*Merges two sorted lists*/

List Result
List M1,M2 /* Locals so that we do not change the parameters! */

M1 := M1in /* Copy parameter */
M2 := M2in /* Copy second parameter too! */
Result := NewList
while .not. (IsEmpty(M1) .or. IsEmpty(M2))
  do /* Done when one is empty! */
     if First(M1) <= First(M2)
       then /* Take next value from M1 */
          Result := AddFirst(Result,First(M1))
          M1 := DeleteFirst(M1) /* Don't need first number */
       else /* Take from M2! */
          Result := AddFirst(Result,First(M2))
          M2 := DeleteFirst(M2) /* Discard first after copying */
       end
  end

/*One of the two lists is empty - catenate them all together!*/

return(Conc(Reverse(Result),Conc(M1,M2))) /*and reorder Result!*/
```

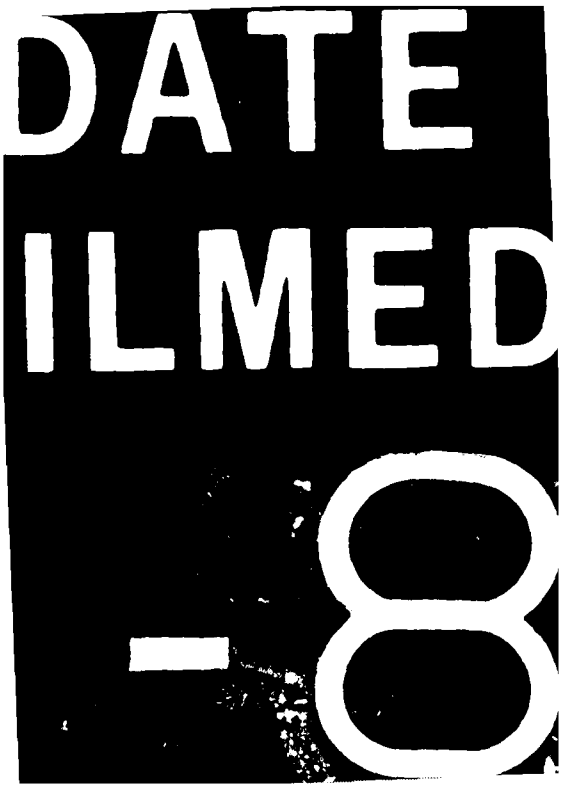
Appendix III The sort routine given to the control group

```
rec List func Sort(List Inlist) /*Sorts into increasing order*/

/* This procedure works by a merge sort - Split the list in */
/* two, sort each half, and then merge the two sorted halfs! */

List Half1,Half2

Half1 := NewList /* Initialize! */
Half2 := Inlist /* Initialize! */
while ListLength(Half1) < ListLength(Half2) do
      Half1 := AddFirst(Half1,First(Half2))
      Half2 := DeleteFirst(Half2)
  end
return(Merge(Sort(Half1),Sort(Half2)))
```


DATE
ILMED
-8